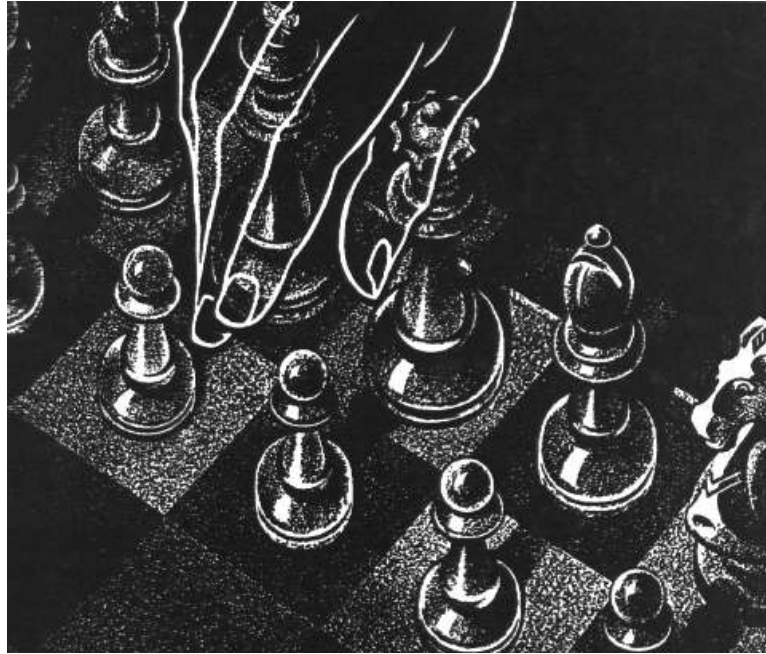


OBJECT-ORIENTED PROGRAMMING CS252

# Programming Assignment



Karolos ANTONIADIS

Panayotis LIARAKIS

## Intro

Chess is a worldwide known board game with millions of fans. It owes most of its popularity to the fact that it's entirely based on tactics and insight, while chance doesn't play any part to its outcome. The game's roots date back to the 10th century in the middle east while different variants have had emerged in several regions of the globe. Its current version, is played on a 64 square board, where each player controls 16 pieces: 8 pawns, 2 Bishops, 2 Knights, 2 Rooks, 1 Queen and a King. The goal of the game is to terminally threaten the enemy king making him incapable of moving. Each piece has a repertoire of moves being able to capture enemy pieces or threaten the enemy king.

## Project

The project assigned for this semester requires to design and implement a chess game using the Java programming language. Our task is to create a one-on-one interactive 2-D game, using both graphical and command line interface. It is crucial that object oriented techniques are used such as polymorphism, inheritance and encapsulation. Besides the previous, obvious goals are also code reusability and modularity. In our project, design has been carefully planned resulting to the least possible redundancy and memory usage while providing a user-friendly interface for a two player match. The game consists of three basic components, the model(responsible for holding the current state of the game and all of the moves), the controller(responsible for checking the validity of the continuance of the game, the previous and next operations, and the implementation of the menu operations) and the view component(supplying the user machine-interaction). The final package also communicates with the model and the controller. Inheritance is used extensively in the model package since the moves classes are very closely related, while the pieces share some common characteristics. Polymorphism is also used in those cases. All members or methods of a class that are meant for usage internally are declared private for encapsulation's sake. Generally visibility modifiers are entirely declared in a "need to know" basis. Finally for security and efficiency any variable that needs to remain unchanged is declared final.

It is required from us to illustrate the game in command line and graphic user interface. Before a game begins the Player needs to choose which type of outline(command line or GUI) he wants.

Command line interface is required to support a series of operations which are selected by a menu interface with the following options:

1. New game

Creates a new game and initializes it with the correct pieces on a

chessboard. The players are called to enter their names and their pieces' color or the computer can select the color randomly. Also prints the game in the screen.

2. Save  
Saves the current game in a coded form in a file.
3. Load  
Loads a game, by decoding a file with a saved game.
4. Exit  
Exits the game.
5. Previous  
The chessboard goes a move back in time.
6. Next  
The chessboard progresses a move forward.
7. Play  
Given a move in a string format like this: "a2 => a3" executes the move on the chessboard and shows the new board.

The chessboard is illustrated in the console using underscores and pipes and letters corresponding to the appropriate pieces. It will look like this:

```

      |---|---|---|---|---|---|---|---|
  8 | r | k | b | q | w | b | k | r |
      |---|---|---|---|---|---|---|---|
  7 | p | p | p | p | p | p | p | p |
      |---|---|---|---|---|---|---|---|
  6 |   |   |   |   |   |   |   |   |
      |---|---|---|---|---|---|---|---|
  5 |   |   |   |   |   |   |   |   |
      |---|---|---|---|---|---|---|---|
  4 |   |   |   |   |   |   |   |   |
      |---|---|---|---|---|---|---|---|
  3 |   |   |   |   |   |   |   |   |
      |---|---|---|---|---|---|---|---|
  2 | P | P | P | P | P | P | P | P |
      |---|---|---|---|---|---|---|---|
  1 | R | K | B | Q | W | B | K | R |
      |---|---|---|---|---|---|---|---|
      a  b  c  d  e  f  g  h

```

Graphic interface is required to support a series of operations which are selected by a menu interface with the following options:

1. File

File has the following choices:

(a) New

Creates and initializes a new game, illustrating it graphically on a window. The players are called to enter their names and their pieces' color or the computer can select the color randomly.

(b) Save

Saves the current game in a coded form in a file.

(c) Load

Loads a game, by decoding a file with a saved game.

(d) Exit

Exits the game.

2. Edit

Edit has the next selections:

(a) Previous

Shows the board by executing the previous move. If there is no previous move, this option is not applicable.

(b) Next

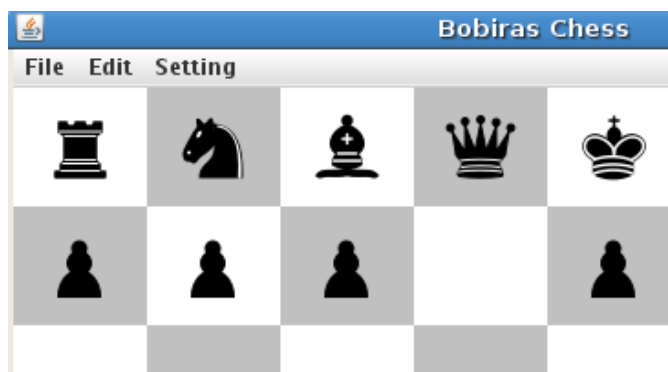
Shows a new board by executing again a previous move. If there is no next move, meaning we haven't gone back, this option is not applicable.

3. Settings

(a) ProposedMoves

If selected, displays all the possible moves of the selected piece.

The GUI could look like this:



## Package model

In this particular package the pieces, chessboard and moves classes are implemented. That means it consists of the classes that are responsible for the current state of the game and the piece's moves. This package contains two sub-packages, the model.moves package and the model.pieces package. The model.pieces package contains the following classes: Piece, Pawn, King, Queen, Rook, Bishop, Knight and the model.moves package contains these classes: Move, Promotion, CapturePromotion, Capture, Castling and PromotionException. The model package just holds Cell and ChessBoard class. Following next are the information regarding the behavior(methods) and the characteristics(attributes) of each class.

### Class Cell

This specific class depicts a square, one of the 64 positions of a chessboard. A square in a chessboard is described by its coordinates(row and column), its color and which piece it holds. So the following are the attributes of this class:

```
private final int x,y;
private Color color;
private Piece piece;
```

Fullfilling the encapsulation goal, the attributes of this class are declared private(same logic is used in all classes). The x, y coordinates are declared final since a square's position is immutable(the use of squares is clarified in the ChessBoard class). Attribute color reflects the color of the particular square which initially is either black or white. We may need an extra color, for instance when implementing proposed moves. The attribute piece holds the type of piece that is stored and "occupies" the particular square. If a square is empty, piece is null. In a chessboard's box we need to be able to place pieces on it, retrieve it's coordinates in relation with the chessboard, change its color, et cetera. So these are the Cell class methods:

1. *public Color getColor(); accessor(selector)*  
Returns the color of this particular Cell.
2. *public void setColor(Color x); transformer(mutative)*  
Changes the color of this particular Cell.
3. *public Piece getPiece(); accessor(selector)*  
Returns the Piece that this specific Cell contains. If the Cell is empty, null is returned.
4. *public void setPiece(Piece x); transformer(mutative)*  
Changes or places a Piece in this particular cell.

5. *public int getX(); accessor(selector)*  
Returns the ordinate of this Cell
6. *public int getY(); accessor(selector)*  
Returns the abscissa of this Cell

#### Class ChessBoard

This class depicts a chessboard. A chessboard is practically an 8x8 grid which contains 64 black and white squares. This class has only one attribute, the following:

```
private Cell [][] board;
```

which is initialized from ChessBoard constructor in this way:

```
board = new Cell [8][8];
```

That means, a chessboard is a two-dimensional array that consists of boxes(Cell). This class has the following methods:

1. *public Cell getCell(int x, int y); accessor(selector)*  
Returns the chessboard Cell which is defined by the parameter coordinates. x references row, while y references column of the board.
2. *public void setCell(int x, int y, Cell z); transformer(mutative)*  
Sets the Cell defined by the coordinates to the one passed as parameter.
3. *String toString(); accessor(selector)*  
Creates a string representation of the board. This method is used in the CommandLineUI.

#### Class Piece

Class piece is declared as abstract and depicts a piece. This class is extended by the following classes:

1. Pawn
2. King
3. Queen
4. Bishop
5. Knight
6. Rook

Pieces have some shared characteristics, since they all are described by the same color(black or white), a name, a move and an icon. Since there is no point in having a Piece instance, class Piece is declared as abstract. The class contains the following attributes:

```
private final boolean color;
private String name;
private Icon icon;
```

where 'color' is the color and 'name' is the name of the piece(for instance "p" for pawn). The color of a piece can be white(for true) or black(for false) and cannot change during a game, that's why it's declared boolean and it's final. 'icon' is the picture of the piece. This class implements the following methods:

1. *public boolean getColor(); accessor(selector)*  
Returns the color of a specific piece. There is no setColor(), because obviously a piece can't change color during the match. Therefore a piece's color is acquired during its instantiation.
2. *public String getName(); accessor(selector)*  
Returns the the name of the given piece.
3. *public void setName(String x); transformer(mutative)*  
Changes the name of the given piece with x.
4. *protected List <Move> verticalMoves(Cell x, ChessBoard z, boolean isKing);*  
Given a cell and a chessboard this method calculates all the possible vertical moves (meaning same column) the piece from Cell x can perform, adds them to a list and returns them.
5. *protected List <Move> horizontalMoves(Cell x, Chessboard z, boolean isKing);*  
Similar to verticalMoves, but instead of vertical moves, it calculates all the horizontal moves(same row) and returns them.
6. *protected List <Move> diagonalMoves(Cell x, Chessboard z, boolean isKing);*  
Similar to verticalMoves, but instead of vertical it calculates diagonal moves.

7. *public Icon getIcon(); accessor(selector)*

Returns the Icon of the given piece.

8. *public void setIcon(Icon r); transformer(mutative)*

Changes the Icon of the given piece with one passed as parameter.

Methods 4,5 and 6(they are protected since they will not be used outside the model.pieces package) are implemented in Piece, so that the King, Rook, Bishop and Queen classes have them ready for usage, since a rook can move vertically, as can the king and queen. However the king can only move by one square(diagonally, vertically or horizontally), that's why we pass an isKing boolean parameter on those methods. This way when isKing is true we find all the moves that are possible in x's neighbouring cells. These methods spare us from excessive code. Also class Piece contains the following method:

```
public abstract List <Move> moves(Cell x, ChessBoard z);
```

This class is implemented by all of Piece's subclasses(as it is abstract), and returns a list of all the possible moves of the piece in this particular Cell.

e.g in the method moves() of Queen's Class we just execute the following:

```
a = horizontalMoves()  
b = verticalMoves()  
c = diagonalMoves()
```

and simply return the union of the lists a, b and c.

Also all the subclasses of the Piece class override the toString() method which simply returns a String based on the piece's name. Additionally, the King and Rook contain an extra attribute which is:

```
private boolean hasMoved;
```

and provides us the knowledge whether this piece has moved at least once(that means in instantiation it is initialized as false, and when a move is executed it becomes true) so that we can know if a castling move is permitted. Also they have two extra methods which are:

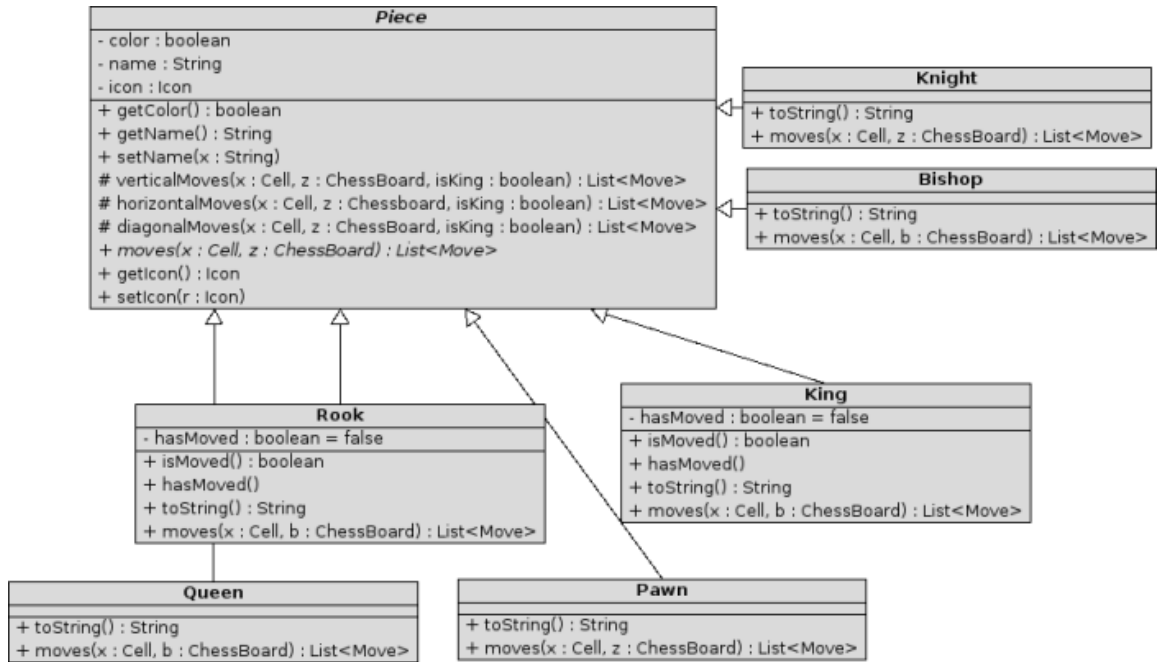
1. *public boolean isMoved(); accessor(selector)*

Returns a boolean value showing us whether the King has moved, returning true if so, false otherwise. This value is used to determine if the Castling Move is permitted.



2. *public void hasMoved(); transformer (mutative)*  
Sets the hasMoved attribute of Class King to true, indicating that the King has moved. If so the Castling Move is now unpermitted.

The UML class diagram of Piece and it's subclasses is:



All the Piece subclasses, inherit it's methods. In the UML class diagram, the subclasses have only the methods that are being overridden from the base class (the inherited methods of the Piece class that are not overridden are not shown, but they exist).

### Class Move

This class defines a move. A move is characterized by its source and destination coordinates. Therefore this class has two attributes which are both of type Cell:

```
private Cell from, to;
```

They indicate where you were, and where you have gone. Move's constructor is declared as so:

```
public Move(Cell a, Cell b) {
    from = a;
    to = b;
}
```

}

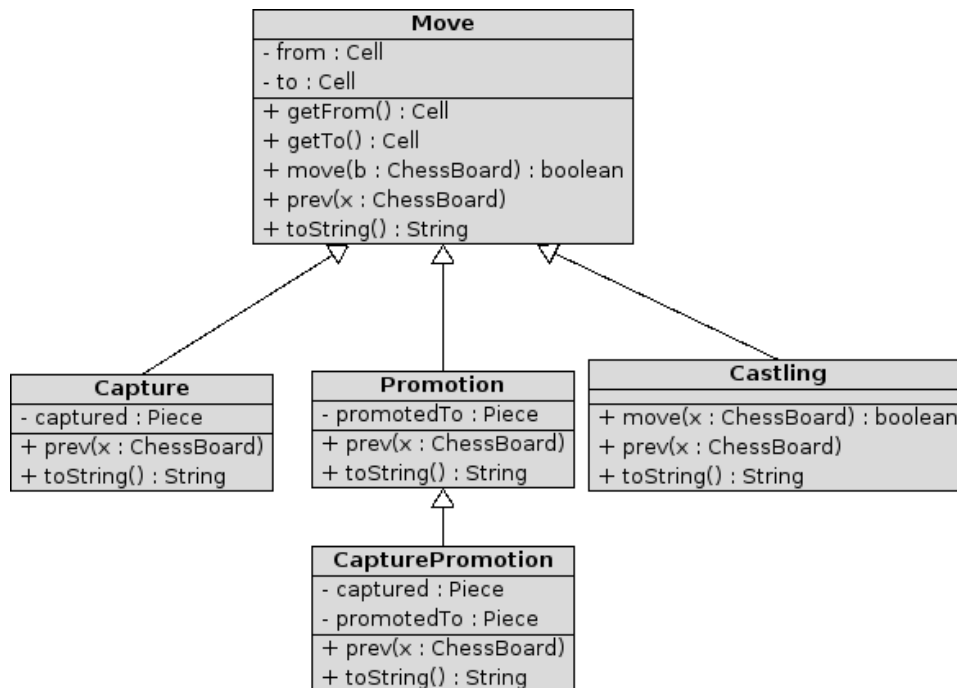
The class contains the following methods:

1. *public Cell getFrom(); accessor(selector)*  
Returns the source Cell. This Cell is the same with the one passed as first parameter in Move's constructor, therefore any changes to this Cell are also applied on the latter.
2. *public Cell getTo(); accessor(selector)*  
Returns the destination Cell. This Cell is the same with the one passed as second parameter in Move's constructor, therefore any changes to this Cell are also applied on the latter.
3. *public boolean move(ChessBoard x);*  
Applies the move on the parameter ChessBoard. Returns true if the move is executed, false otherwise.
4. *public void prev(ChessBoard x);*  
Applies the reverse move. ChessBoard(goes from 'to' to 'from'). Theoretically this move is always possible when called upon. (More info in class Control of the controller package). It's used for Previous and Next operations of the game.
5. *public String toString(); accessor(selector)*  
Returns the String representation of the given move. It's mostly used in the Save and Load operations.

Class Move has the following subclasses:

1. Capture
2. Promotion
3. Castling

because all of them are special type of moves. Promotion has also a subclass with the name: "CapturePromotion". Their relationship is displayed below in the UML diagram:



Again, in the subclasses of Move, only the methods that override Move methods are shown.

Class Capture is of Move type with the extra ability of captivating an opponent's piece. Castling is the small and great castling move(what else could it be?). Promotion is the move where a pawn that reaches the last line of the Chessboard can convert to a queen, knight, bishop or rook. CapturePromotion is similar with Promotion, with the ability to simultaneously captivate an opponent's piece. CapturePromotion extends Promotion instead of class Capture because the prev() of CapturePromotion have more similarities with Promotion than with Capture. Both Capture and CapturePromotion class contain an extra field:

```
private Piece captured;
```

in order to be able to successfully implement the prev() method of these two classes. Castling is the move when a player moves his King two squares to the right or left(under certain circumstances). Also the Promotion and CapturePromotion classes contain one more field:

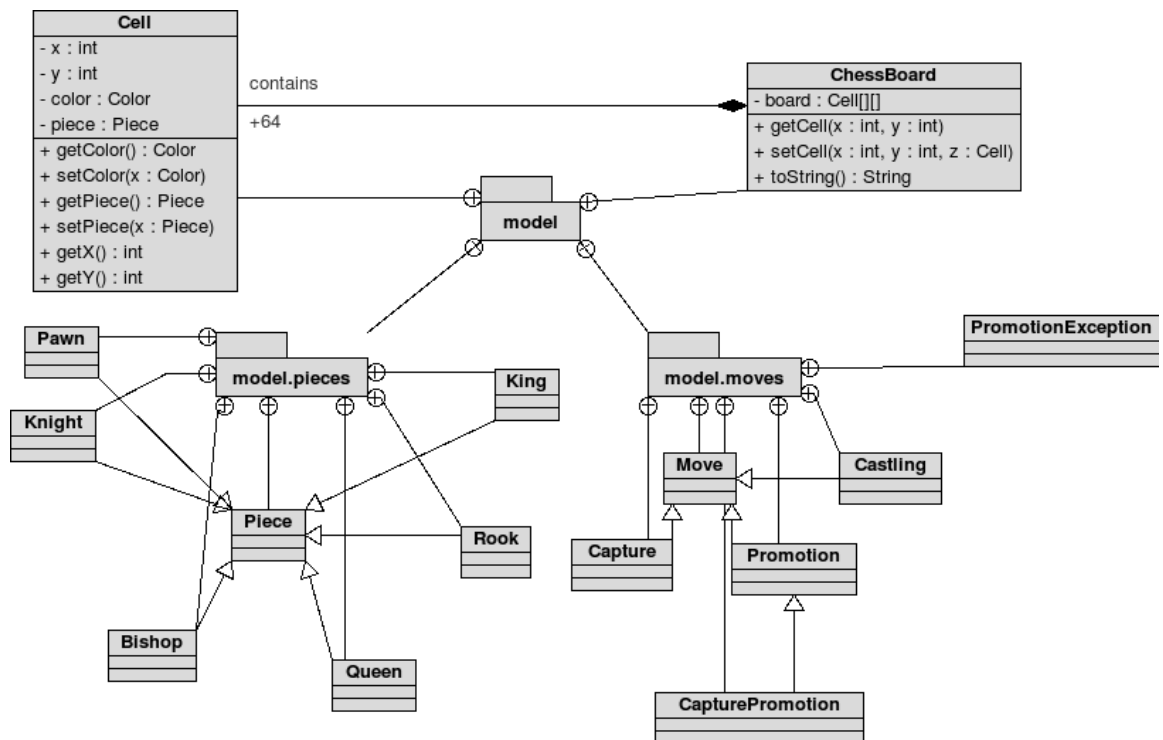
```
private Piece promotedTo;
```

again, to be able to implement their prev() method.

Class PromotionException

This class extends the RuntimeException and is used in the updateGame method of the Control class of the controller package. It is located here since it's more associated with this package, both are connected to class Move.

Here is a UML class diagram concerning the entire model package:



## Package controller

In this particular package Control and Player class are implemented. The Control class is the "brain" of the game and is responsible for creating a new game, loading and saving a game, checking whether the match is over(draw or checkmate), observes whether the king is threatened(check) and also is responsible for executing previous and next moves. Next up are information concerning the behavior(methods) and characteristics(attributes) of the Player and Control class:

### Class Player

This particular class describes and defines a player based on his name, the color of his pieces (true for white, false for black) and if he has clearance to play. Therefore it's attributes are the following:

```

private final String name;
private boolean canPlay;
private final boolean color;

```

The variable color is declared as boolean(true for white, false for black), since we think it will simplify and optimize the implementation of various functions. Also it's declared as final due to the fact that the color acquired during initialization is retained throughout a match. String name is the name of the Player, it's also final since it can't change during a match. Boolean attribute canPlay shows us whether it's the player's turn or not. Player's turns succeed each other. Class Player implements several methods regarding a turn of a player and its characteristics. Here are the methods of the Player class:

1. *public boolean getColor(); accessor(selector)*  
Returns the boolean value indicating the color of the pieces of a player (true for white, false for black).
2. *public void setTurn(boolean play); transformer(mutative)*  
Sets the canPlay attribute of Player to the parameter of play value.
3. *public boolean getTurn(); accessor(selector)*  
Returns the value of canPlay, indicating whether it's a player's turn.
4. *public String getName(); accessor(selector)*  
Returns the value of the players name.

#### Class Control

This class is responsible for creating a new game since it instantiates the chessboard. It has 4 attributes, 2 lists of previous and next moves, a chessboard, and a boolean value that indicates whether the match can continue. So the attributes are:

```

private ChessBoard board;
private List<Move> previousMoves , nextMoves;
private boolean canPlay;
private Player first , second;

```

ChessBoard board is the board that all the operations are performed. The list previousMoves holds all the moves that have been executed, and in case we go back, nextMoves holds the moves that can be executed. If no moves have been executed both lists are empty, if our state is the current state of the chessboard, nextMoves is empty, and if we go back to our initial state previousMoves is empty. Boolean attribute canPlay is determined on whether we are at the current state or we have gone back at least one move, and defines if we can play. If at least one reverse move has been executed,

canPlay is false until we reach again the current state. The attributes first and second corresponds to the players of the current game.

For control we need a save and load game method and new game method, a series of methods checking the chessboard's state and the methods that implement the previous and next operations:

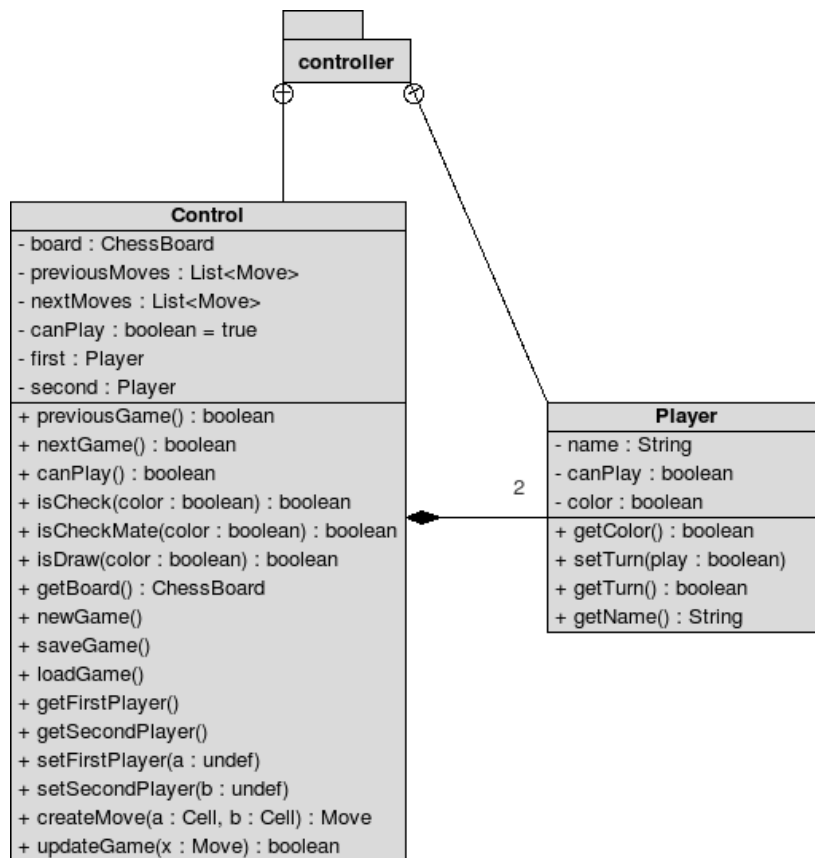
1. *public boolean previousGame(); transformer(mutative)*  
Implements the previous move operation popping every move from previousMoves list, executing the Move and then pushing it to the nextMoves list. Returns a boolean value notifying whether the operation is successfull. True if it is, false otherwise.
2. *public boolean nextGame(); transformer(mutative)*  
Implements the next move operation, popping moves from the nextMoves list, executing them and pushing them to previousMoves list. When nextMoves list is empty canPlay converts to true. Returns a boolean value whether the operation is successfull, true if it is, false otherwise.
3. *public boolean canPlay(); accesor(selector)*  
Returns the value of the canPlay attribute which indicates whether we can play.
4. *public boolean isCheck(boolean color); accesor(observer)*  
Returns a boolean value depicting king's state, parameter indicates which king. True if king is checked, false otherwise.
5. *public boolean isCheckMate(boolean color); accesor(observer)*  
Returns a boolean value depicting the king's final state, parameter indicates which king. True if king is checked(ending of the game), false otherwise.
6. *public boolean isDraw(boolean color); accesor(observer)*  
If the player with the pieces of the given color cannot move but is not checked, true is returned, meaning that the game is over with a draw, otherwise false.
7. *public ChessBoard getBoard(); accesor(selector)*  
Returns the current chessboard.

8. *public boolean updateGame(Move x); transformer(mutative)*  
 Updates the chessboard executing the move passed as parameter. If the given Move is instance of Promotion throws an PromotionException(which is located in the model.moves package) which means that there has been a pawn that needs to be promoted and the player is required to select what this pawn would be promoted to.
9. *public void newGame(); transformer(mutative)*  
 Creates a new game and initializes the pieces in the appropriate positions.
10. *public void saveGame(); accessor(selector)*  
 Saves the current match in a file in an encoded form.
11. *public void loadGame(); transformer(mutative)*  
 Loads and decodes from a file data, in order to create a new game based on that data.
12. *public Player getFirstPlayer(); accessor(selector)*  
 Returns the attribute first.
13. *public Player getSecondPlayer(); accessor(selector)*  
 Returns the attribute second.
14. *public void setFirstPlayer(Player a); transformer(mutative)*  
 Set the first Player to the parameter a.
15. *public void setSecondPlayer(Player b); transformer(mutative)*  
 Set the second Player to the parameter b.
16. *public Move createMove(Cell a, Cell b);*  
 Based on the parameter Cell(s) decided what type of Move it is(Capture, Promotion, ...), then creates the appropriate Move and returns it. Used when the user gives 2 Cell(s)(either on command line or GUI) to know what kind of Move these Cell(s) correspond to.

This class is the brain of the game, since after every turn methods isCheckMate(), isDraw() decide the end or not of the game. isCheck() is

used to verify all the permittable moves of a piece or to notify us whether the king is checked. The other methods alter the chessboard, either by updating it or by accessing it. Also the methods used for the next and previous operations are implemented here and are contained in the menu operations. Finally this class arbitrates whose turn it is to play.

Here is the UML class diagram concerning the controller package:



### Package view

In this specific package, all the UI classes are implemented. It is divided into 2 sub-packages, the view.cl package which contains methods connected to the command line interface and the model.gui package which contains classes that deal with the Graphic User Interface. The view.cl package consists of a CommandLineUI class which sketches the game in command line using string representations and the view.gui package contains the GraphicUI class which creates a visual image of the game. The view package has the UserInterface interface which is implemented by both Command-



LineUI and GraphicUI class and defines the common methods of the two user interfaces and the IllegalViewException class. The behaviour and the characteristics of each class are described below.

#### Interface UserInterface

Interface UserInterface holds the declarations of basic methods that are both used from class CommandLineUI and GraphicUI but are implemented differently. The interface holds the following methods (more details are shown in CommandLineUI and GraphicUI class report):

1. *void showGame();*
2. *boolean makeMove(Cell source, Cell dest);*
3. *void nextGame();*
4. *void previousGame();*

#### Class CommandLineUI

Class CommandLine is responsible for the graphic depiction of a chess game in command line. This class implements the UserInterface and has one attribute is, which is:

```
private Control game;
```

At startup when we choose the way we want to see the game (in command line or in a graphic interface) we create an instance of the CommandLineUI class for the appropriate depiction and in the class we create a new instance of Control thus creating a new match. A game is constructed and initialized as follows:

```
game = new Control();  
game.newGame();
```

Class CommandLineUI implements methods regarding the appearance of the game and its state (whether its over or not), and also the previous and next operations. The methods are described analytically below:

1. *public void showGame(); accessor(selector)*  
Depicts the game on command line using the toString method of the ChessBoard class.
2. *public boolean makeMove(Cell source, Cell dest); transformer(mutative)*  
Executes the move which is provided by the parameter Cell(s). Returns true if move is successfully executed, false otherwise. This method invokes the updateGame method of the Control class (controller

package).

3. *public boolean makeMove(String x); transformer(mutative)*  
Converts the input String to a set of source and destination Cell(s) which corresponds to a Move, and invokes with these two Cell(s) the upper overloaded makeMove method. Returns true if move is successfully executed, false otherwise.
4. *public boolean isOverWin(boolean color); accessor(observer)*  
Returns a boolean value notifying whether the game has ended victoriously for the player with the pieces of the given color. True if game is over, false otherwise. Is called upon after every turn played. In case of a win, game ends.
5. *public boolean isOverDraw(boolean color); accessor(observer)*  
Returns a boolean value notifying that the game has ended with a draw when it's the player's turn with the given piece's color. True if game is a draw, false otherwise. In the first case, the game ends.
6. *public void nextGame(); transformer(mutative)*  
Is used for depicting the next board from the one we are now, if there is such a board. In order for this method to be executed, it is vital that at least one previousGame() has been executed. Invokes the class Move's move() method.
7. *public void previousGame(); transformer(mutative)*  
Is used for depicting the previous board from the current one if there is such a board. In order for this move to be executed at least one move must have been executed. Invokes the Class Move's prev() method.

The constructor of CommandLineUI doesn't take any parameters and asks the user to give the names of the players and the color of each one.

The above functions are used to supervise the command line implementation either using methods such as (showGame) or using game data methods to conclude or continue the game (isOverWin, isOverDraw, makeMove). To check the results of these methods CommandLineUI needs an instance of class Control since the Control methods are implemented in controller package. The methods responsible for depicting the previous and next operations(previousGame, nextGame) use the methods previousGame and nextGame from class Control and throw an IllegalArgumentException(see

IllegalViewException class) if you cannot go back or forth.

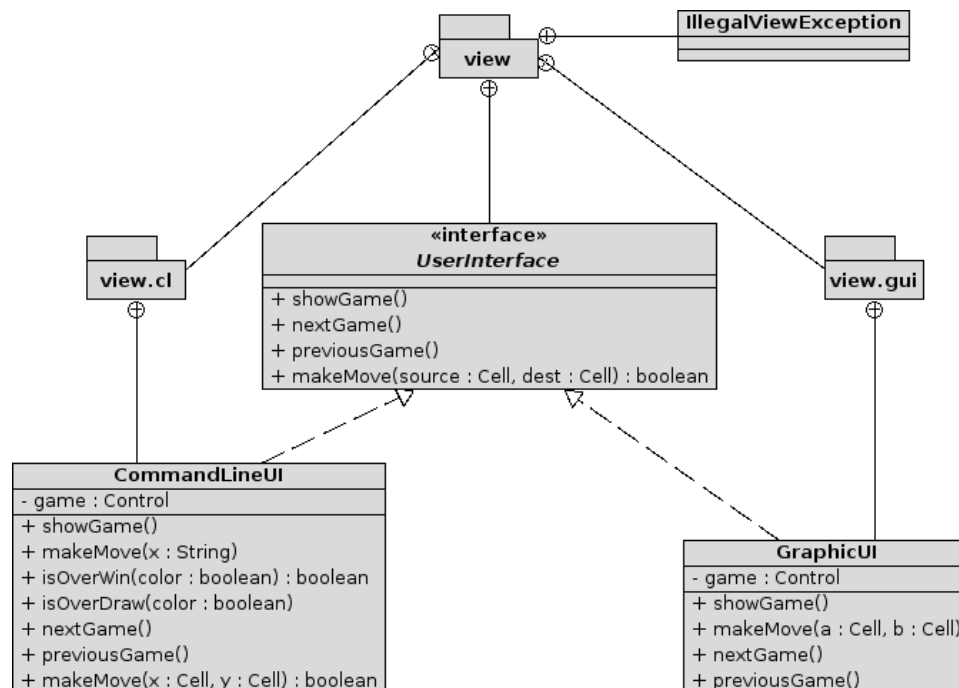
### Class GraphicUI

This class handles the GUI of the chess game, meaning it shows the board and the pieces in a window environment. It's hard to predict the methods and the attributes that this class will contain, of course it will implement the methods declared in the interface `UserInterface`. It's also difficult to foresee the classes that the `view.gui` package will require, since it's our first time designing GUI applications.

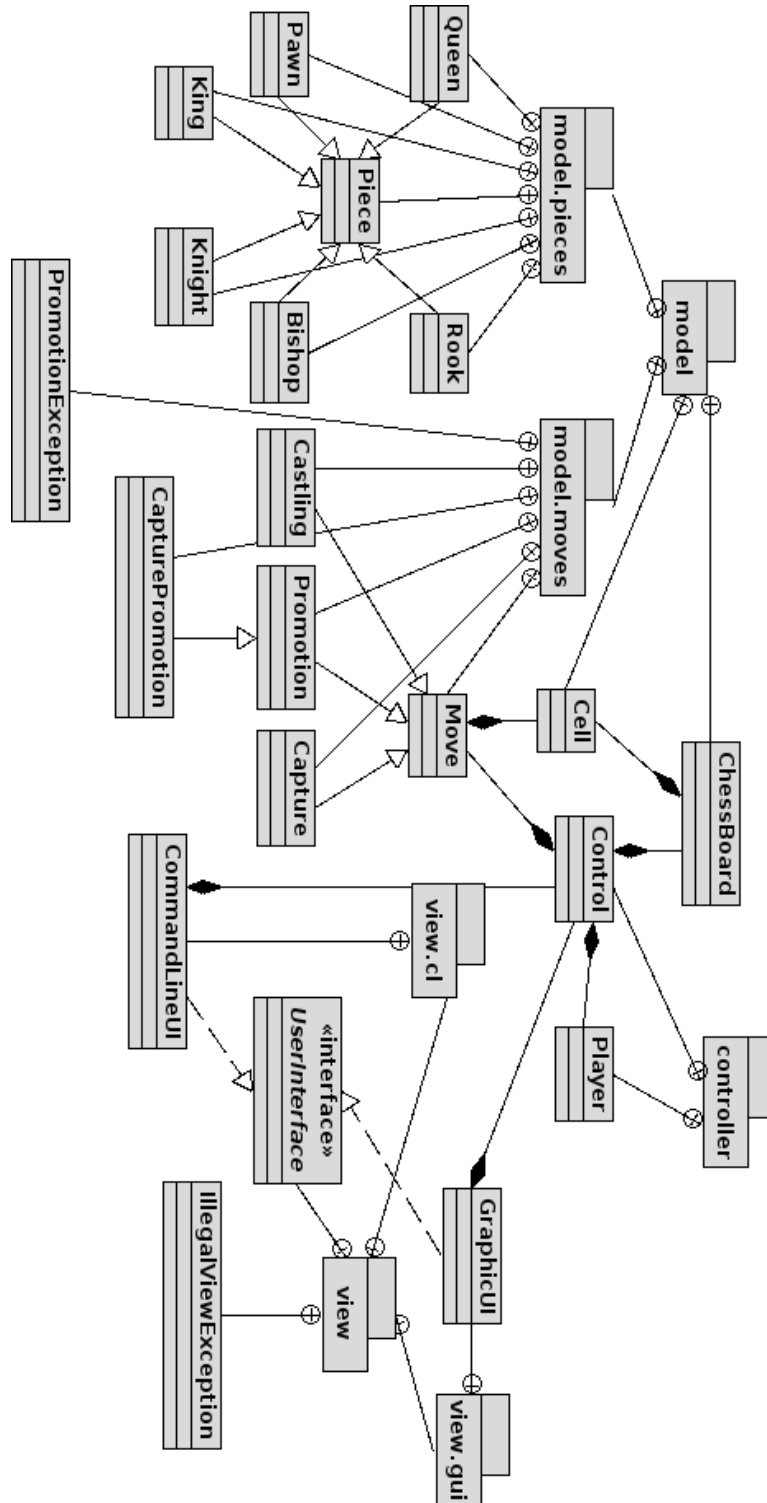
### Class IllegalViewException

Class `IllegalViewException` is a runtime exception that is thrown from the functions `nextGame()` and `previousGame()` if there is no previous or next board to display. It extends the `RuntimeException` class and is used for indicating whether the operation requested is permitted.

Here is the UML class diagram concerning the view package:



### UML Class Diagram for all packages and classes



### **Where does the game start?**

Aside all the above classes and packages, a Main class is needed where the game will begin. This class will be in the main package and will contain a main method where by choice of the user an interface is picked and the game will begin.

*(\* Some of the above methods may throw Exceptions of various kinds, those methods and all the others are analytically described in the project Java doc's comments)*

## Debriefing Report

### *Changes Applied:*

1. The Rook and King class have been enriched with an (overloaded) public void hasMoved(boolean what) method that alters their hasMoved attribute based on the parameter what. The hasMoved attribute informs us whether the piece(Rook or King) has moved. In order to use the previous operation, the hasMoved attribute of King and Rook need to be changed to their initial state, something that could not be realized in our prototype design. Obviously this is used during the castling processes.
2. The Promotion class has been enlarged with a public void setPromotedTo(Piece what) which assigns the private Piece promotedTo attribute to the Piece passed as parameter. This attribute and method is used in the Promotion operation, so as to acknowledge the Piece in which the promoted Piece is going to be promoted.

### *Algorithms Used:*

1. The public boolean isCheck(boolean color) method uses a very simple algorithm. Depending on the parameter color it calculates and sums up all the moves of the opponent's pieces' Moves and simply check's if one of a Move's destination Cell is the where the King lies. In such case the King is under threat.
2. The public boolean isCheckMate(boolean color) method checks whether your king is under threat. If true it finds and sums up all your Piece's Moves. If any of these Moves places the King in a non check state, then the King is not terminally checked. Else the game ends with a Check-Mate.
3. The public void saveGame() implements the save operation and uses the following algorithm: It opens a text file, initially writing the names of the players and after that it writes all the Moves executed until the current point of the game.
4. public void loadGame() implements the load operation and uses the following technique: It opens the text file, where the game has been saved, and creates a new Game based on the players written on that file. After that it executes all the Moves recorded on that file concluding to the desired state of game.

**THE END**